Part IV

INSTRUMENTATION AND MONITORING

Introduction

By their very nature, Grid environments are highly distributed and composed of many elements. The goal of the Grid is to hide much of the complexity of the underlying system from the user. This is great when everything is working fine, but when there are problems the Grid user needs a means of seeing precisely what is happening. Also, with the range of resources available, Grid processes need a way to understand the current status of various elements in order to make resource allocation decisions.

Grid Monitoring is the measurement and publication of the state of a Grid component at a particular point in time. To be effective, monitoring must be "end-to-end", meaning that all components between the Grid application endpoints must be monitored. This includes software (e.g., applications, services, middleware, operating systems), end-host hardware (e.g., CPUs, disks, memory, network interface), and networks (e.g., routers, switches, or end-to-end paths). Instrumentation is the process of adding probes to generate monitoring event data.

Grid Monitoring is required for a number of purposes, including status checking, troubleshooting, performance tuning, and debugging. For example, assume a job has been submitted to a Grid Resource Broker, which users a Grid Reliable File Transfer Service to copy several files to the site where the job will run, and then runs the job. This particular process should normally take 15 minutes to complete, but two hours have passed and the job has not yet completed. Determining what, if anything, is wrong is difficult and requires a great deal of monitoring data. Is the job still running or did one of the software components crash? Is the network particularly congested? Is the CPU particularly loaded? Is there a disk problem? Was a software library containing a bug installed somewhere? Monitoring provides the information to help track down the current status of the job and locate any problems.

This chapter provides background on the challenges of instrumentation and monitoring in Grid systems. We first describe in more detail what Grid components need monitoring data, and why. We then describe the basic pieces of a Grid monitoring system necessary to move monitoring data from where it was measured to the component that needs it. A survey of monitoring tools and frameworks is presented, including Dyninst, GMA, Network Weather Service, NetLogger, Ganglia, MDS, OGSA-based monitoring, as well as commercial systems such as SNMP and CIM. Finally, we present some detailed case studies that demonstrate how the various components interact to solve a particular problem.

Consumers of Monitoring Data

Many Grid middleware components need instrumentation and monitoring data. For example, a Grid Scheduler, or *Superscheduler*, as described in chapter N on Grid Resource Management, is an example of a Grid middleware component that requires monitoring data in order to make intelligent scheduling decisions. Although jobs may be assigned to compute resources based solely on relatively static data such as CPU capability, total memory, operating system, and network capacity, better optimization may come from incorporating dynamic monitoring data such as batch queue length, CPU or network load, or local storage system load.

Network monitoring data is particularly important to Grid middleware such as the Replica Manager described in chapter N. Selecting the best source to copy the data from requires a prediction of future end-to-end path characteristics between the destination and each possible source. Accurate predictions of the performance obtainable from each source requires measurement of available bandwidth (both end-to-end and hop-by-hop), latency, loss, and other characteristics important to file transfer performance.

Other services that require monitoring data include the security and accounting services described in the next chapter, and the application tuning mechanisms described in chapter N.

In general, the purpose of collecting monitoring data is for one or more of the following:

- *Troubleshooting and Fault Detection*: When a Grid job fails, it can be very difficult to determine the source of the failure. Comprehensive monitoring information is needed for both real-time fault detection and for post mortem analysis.
- *Performance analysis and tuning*: Developers Grid applications and middleware often observe performance problems such as unexpectedly low throughput or high latency. Determining the source of the performance problems requires detailed end-to-end instrumentation of all components, including the applications, operating systems, hosts, and networks.
- *Guiding scheduling decisions*: In a grid, the available communication and computational resources constantly change. Measurements of the current state of the grid are used to drive resource allocation and scheduling decisions. Chapter N discusses how to use this type of performance data.
- *Gathering data to improve the next execution of the program*: Historically, this has been the primary use of performance instrumentation. Data is gathered during an execution and used by the programmer or the compiler to alter the program so that it will run faster when next executed. Chapter N discusses how to present performance data to application programmers, and Chapter N describes compiler use of performance data.
- *Adapting a computation while it is in execution*: For long running programs in a grid, the computational resources required or the available resources might change during a single execution. Performance data can be used to either permit computational steering (described in Chapter N) or to permit applications or runtime libraries to automatically change in response to these observations such as the Active Harmony System described in this chapter or the Autopilot system described in Chapter N.
- *Debugging*: Complex, multithreaded, distributed programs can often be difficult to debug. The proper instrumentation and analysis tools can aid the debugging process.

End-to-End Monitoring Components

A complete end-to-end monitoring system has several components, as shown in Figure 1.

- *Instrumentation*: Instrumentation is the process of putting probes into software or hardware to measure the state of a hardware component, such as a host, disk, network, or a software component, such as operating system, middleware, or application. These probes are often called *Sensors*. Facilities for precision instrumentation of Grid applications and middleware and hardware resources are essential to the process of tuning and debugging. It is common for Grid applications to run much slower than expected, and without detailed instrumentation data, it is nearly impossible to determine the source of the bottlenecks. Additionally, it can be extremely difficult to debug deadlocked threaded, distributed applications without detailed instrumentation.

- *Monitoring Event Publication*: Consumers of monitoring event data need to locate appropriate monitoring event providers. Standard schemas, publication mechanisms, and access policies for monitoring event data are required.
- *Event Archives*: Archived monitoring event data is critical for performance analysis and tuning, as well as for accounting. Historical data can be used to establish a baseline upon which to compare current performance.

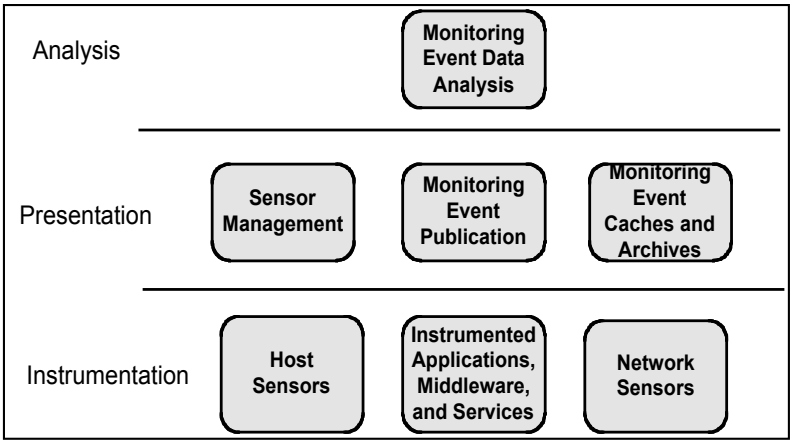

Figure 1: Monitoring Components

- *Sensor Management*: As Grids environments become bigger and more complex, there are more components to monitor and manage, and the amount of monitoring data produced by this effort can quickly become overwhelming. Some components require constant monitoring, while others only are monitored on demand. A mechanism for activating sensors on demand is required.
- *Data Analysis:* Collecting large amounts of monitoring data is useless unless this data answers the user's questions about their application. Data analysis techniques provide a way to reduce raw data to the salient information needed. These can include statistical clustering techniques, data mining, critical path analysis, or other approaches.

Each of these components is discussed in detail in the following sections.

Most of these monitoring components have existed in one form or another in traditional computing environments. However, there are several issues that make the problem considerably more difficult in a Grid environment. A Grid monitoring service has the following requirements:

- Discovery: The distributed nature of Grid systems makes it difficult to discover what types of components are available and how they are performing. Consumers must be able to find and understand monitoring data. This may involve doing distributed queries over multiple registries of monitoring data sources. Common monitoring event descriptions are required.
- Authorization: A Grid monitoring data publication systems must have a mechanism for controlling who has access to what data. For example, some organizations do not want to publish what version of the operating system a host is running, as that makes it easier to exploit security holes. ISPs never publish routing information, even though Grid middleware could use this information to make scheduling decisions. Monitoring events generated by detailed instrumentation of a Grid middleware component may only be useful to developers, and allowing other access to this data may put unnecessary load on the monitoring system.
- Interoperability: A large number of groups have written a wide assortment of monitoring event producers. For this data to be useful on the Grid, monitoring event consumers must be able to understand the contents of that data. Common event formats and types are required.
- Intrusiveness: When monitoring any resource, one must be careful that the act of doing the monitoring does not affect what is being monitored. Intrusiveness has always been critical to instrumentation. However, since a Grid environment includes many shared resources, it

is important to ensure that multiple users monitoring system do not perturb the system being measured or each other's measurements. For example, running too many CPU or network probes will clearly affect the results.

To the best of our knowledge, no current Grid monitoring service actually satisfies all these requirements. Within the Grid research community, a consensus on these requirements has been reached, and several organizations are working on services to address these requirements.

Monitoring Event Data Representation

The form of the collected performance data has an important effect on the time efficiency of instrumentation, space needed for storage, and the type of analysis that can be performed. Instrumentation data can be divided into finite size structures (whose size does not depend on the length of program execution), and trace streams. The finite size structures include counters and timers that are associated with program structures, such as a counter for each procedure or process. These counters or timers are updated at strategic points in the program's execution. Trace streams generate a new datum each time the program executes a traced operation.

Scalars: Counters and Timers

The counter is the simplest form of performance data. It is incremented by an appropriate value at appropriate points in execution. Counters may be allocated for software monitoring events, such as procedure calls, or hardware events, such a cache misses. The main efficiency issue associated with counters is the size: a small counter (32 bit) is currently large enough for most software-based operations and can be updated efficiently on current processors. Large counters (64 bit) are needed for low-level and hardware operations that occur at close to processor clock frequency. Unfortunately, not all current processors can efficiently operate on 64 bit integers and as a result must use a sequence of 32 bit operations.

Timers are a deceptively complex issue. The basic and obvious concerns are that they be precise (clock cycle resolution) and fast to access. New processor designs address this issue. Equally important are two additional abilities: providing both *processor time* (CPU time of a process or thread) clocks and *wall time* (elapsed time) clocks; and virtualizing clocks, to associate them with individual program or system components.

Process and wall time clocks measure fundamentally different activities. Computation time is more easily tracked by process time clocks. Wall time is important for measuring such quantities as I/O or synchronization blocking times. If a processor is dedicated to a single thread or process, only one type of clock is needed, but in modern systems with multiple processors or threads, both types of clocks are important. Most operating systems give fast access to cycle counting wall timers but give slow (system call speed) access to coarse-resolution (often 10 ms!) process timers.

Clock virtualization can be provided in either hardware or software. Hardware virtual clocks are like wall clocks, except that during a context switch the clock value for the old thread is saved into memory and value for the thread to be dispatched is loaded into the clock. A software implementation that stores the wall time at dispatch and the total time consumed by previous dispatches of the process is also possible. However, for the clock to meet the low latency requirement, these values must be stored in memory locations readable by a user process. This software approach requires addition and subtraction operations to compute the virtual time. It is easy to implement cheap, accurate process timers, but operating system vendors continue not to do so.

Traces

The most general form of performance data is the trace. At selected points in a program's execution, data can be added to a trace stream. This data can include any type of information, but is typically split into header information (event type, time stamps, size) and event type-dependent information (such as the number of bytes sent for a message-send event trace).

While tracing is flexible, the size of the data collected can be problematic because the trace size grows with the length of program execution. The desire for detailed (fine granularity) traces increases the frequency of traced events and the desire to measure long-running programs means increasing the duration of tracing. The combination of these two issues can strain disk sizes or network connections.

PICL [1] was an early and influential tracing package that produced a common (and predetermined) format for trace records. The common format allowed many data analysis and visualization projects to share tools and performance data.

The Pablo tracing system [2] increased the flexibility of tracing by including meta data about the trace format as part of the trace log. This Self Describing Data Format (SDDF) gave programmers and tool writers the flexibility to easily add new trace types and include new types of performance variables.

Vectors: Time Series

A compromise between scalar performance data and traces is a time-vector of performance data. The Paradyn *time histogram* introduced this idea to performance tools [3]. The time histogram allows for profiling performance behaviour over a period of time, yet maintains a finite size.

A time histogram is a fixed-size array whose elements store values of a performance metric for successive time intervals. Two parameters determine the granularity of the data stored in time histograms: initial bucket width (timer interval) and number of buckets. Both parameters can be adjusted to meet the need of the application. If program execution time exceeds the initial bucket width times the number of buckets, new space is freed to store additional data by doubling the bucket width and re-bucketing the old data. The longer a program runs, the coarser the granularity of the performance data. For a time interval that needs precise data, a programmer could start a new time histogram for that time interval.

Instrumentation

Instrumentation is the process of putting probes into software or hardware to record statistics. Unless the correct information is gathered at the lowest levels, it is impossible for higher-level software to perform its analysis. This section starts by describing very low-level instrumentation (clocks and hardware event counters) and then discusses instrumentation data representation. It also includes information about several existing data collection architectures.

Clocks

A crucial hardware feature for instrumentation is an accurate clock. The quality of the clocks provided by the system can determine if it is possible to build to some types of instrumentation. There are three aspects of the quality of the supplied clock that affect an instrumentation system: high resolution, low latency access, and multi-node synchronization.

To be a useful measure of fine-grained monitoring events, a clock should provide resolution that is approximately the same as the underlying processor clock. This clock must provide enough

bits so that it does not roll over in a short amount of time. At least 64 bits are required for current processors. Many recent micro-processors, such as the Intel Pentium [4], SPARC v9, DEC Alpha [5] and Itanium, include high resolution clocks.

High resolution alone is not sufficient; for a clock to be useful, it must be accessible with low latency to permit the measurement of fine-grained events. Clock operations need to be supported by user-level instructions that execute with similar performance as register-to-register instructions. The Intel Pentium family provides a clock that meets this requirement; however, SPARC v9 does not.

Clock synchronization is an important issue for building instrumentation for a Grid. Single system multi-processors provide hardware for clock synchronization, however this hardware is generally not available for wide-area systems. Two alternatives are available for wide-area computing. First, the communication network between nodes can be used to synchronize the clocks. Protocols such as NTP provide a way to synchronize clocks to within the variation between packet arrivals on the network, about 1 ms for LAN's or 10ms for WAN's [6]. NTP uses a hierarchy of time servers to distribute clock updates. The top level (called stratum-1) clocks are synchronized to a variety of highly accurate time sources: radio clocks (e.g., WWV), atomic clocks, or Global Positioning System (GPS).

Periodically, each node being synchronized by NTP contacts an NTP time server. By estimating the round trip delay between the time server and the node, the NTP algorithm is able to compensate for the time it takes a time-stamped response to travel from the server to the client.

A second alternative is to use an external time base to synchronize clocks. For example, a highly accurate clock is available from Global Positioning Systems (GPS) satellite receivers. Using GPS it is possible to synchronize clocks to within a few microseconds [6].

Timestamps

In a Grid environment, it is important to have high-precision accurate timestamps as part of each monitoring event. Timestamps are simply the current value of a wall clock. In most parallel and distributed systems, timestamps are not used as the primary source of debugging information since it is not generally possible to get the sufficiently accurate and synchronized wall clocks to uniquely order high frequency events. However, the coarser granularity of events across a Grid may make it possible to use timestamps rather than more sophisticated techniques such as Lamport's logical clocks[7] for debugging.

Hardware Counters

To fully understand Grid applications, it is necessary to understand the behavior of the code running on each node. As more and more features are integrated onto a single chip, it is increasingly important that instrumentation be incorporated into the chips since many useful types of information are no longer visible to an external instrumentation system. Modern processors, such as the Sun UltraSPARC III, Intel Pentium IV, and IBM Power4 provide a rich collection of performance data. For example, the Pentium provides access to its performance data through the Model Specific Registers (MSR's). These registers include a 64 bit cycle clock, and counts of memory read/writes, L1 cache misses, branches, pipeline flushes, instructions executed, pipeline stalls, misaligned memory accesses, bus locked cycles, and interrupts. The UltraSPARC III and Power 4 provide a similar set of counters. In addition to processor counters, networking devices such as switches, routers, and line cards often contain hardware counters that can be moinitored as part of a overall Grid monitoring environment.

A weakness of several current architectures is that their performance registers are protected and accessible only in kernel mode. This restriction means that instrumentation executing in an application program cannot access these registers without an expensive trap into the operating system kernel. Due to these traps, an instrumentation system must collect coarser grained information to amortize the trap overhead over a sufficiently large code region.

Use of hardware counters by applications (and even tools) has been limited due to the difficulty of accessing the counters and the different types of counters that have been provided on different platforms. In fact, even two different versions of a processor chip will have somewhat different counters. The PAPI [8] package from the University of Tennessee helps this situation by providing a single API for accessing hardware performance counters on many different types of platforms. PAPI even provides a way to ask for counters in an abstract way (i.e. Floating Point Operations per Second) and get the appropriate machine counter. However, PAPI is limited by the fact that different hardware companies count events differently. For example, some companies count an FMA (Floating point multiply and add) as one floating-point operation, and others count it as two. These subtle differences are particularly frustrating in a Grid environment where a single application may be running on multiple different types of hardware at once.

Overhead

One of the most important things about data collection is not what is collected, but what is the impact of collecting it. Unless the overhead of instrumentation is kept sufficiently low, the underlying system being measured will be altered by the measurement overhead. The acceptable overhead for an application varies depending on the type of data being gathered and its use. For example, if the data is being gathered for a detailed simulation study, a slowdown of an order of magnitude may be acceptable. However, if the purpose of monitoring is to provide data to a Grid scheduler, an overhead of less than 1% might be required. Somewhere in the middle would be monitoring an application to tune it. In this case, an overhead of 5-15% might be acceptable.

One approach to instrumentation perturbation is to try to subtract the time required for the intrusion. An example is the PREFACE system [9], which assumes programs to be composed of serial and parallel portions. PREFACE instruments the serial portions where the time perturbations can be subtracted without changing event orderings. In the parallel regions, the time necessary for execution of the instrumentation is also subtracted. However, no attempt is made to identify when intrusion introduces changes in the ordering of events. As a result, the approach is not useful in the general case.

Maloney and Reed [10] model the delays and produce a performance prediction for how the program would behave without instrumentation. They studied the timing delays introduced by intrusion and attempted to quantify changes in the ordering of events. They modeled intrusion as delays and constructively recover the compensated timings. Both *time-based* and *event-based* models were developed to compensate for the temporal changes to the application and the order changes possible through differential delays of processes. They developed approximate models of instrumentation intrusion and then refined them until acceptable predictions of performance were possible.

Lumpp and Casavant applied control theoretic techniques to the perturbation problem in message passing systems [11]. The approach includes requirements on the parallel system, the instrumentation, and the parallel program in order to assure recovery is possible. Their work differs from previous work in that the correctness of the ordering of events is paramount. Given sufficient trace information, it is possible to recover the timings that would have occurred in the absence of the instrumentation.

The Paradyn Parallel Performance tools take a different approach to controlling overhead and the volume of data collected [12]. The unique feature of their approach is that it lets the programmer see and control the overhead introduced by monitoring rather than simply being subjected to it. The cost system is divided into two parts: *predicted cost* and *observed cost*. Predicted cost is computed when an instrumentation request is received, and observed cost while the instrumentation is enabled. Predicted cost is used to decide if the requested data is worth the cost of collection. For example, if the user requests performance data whose predicted cost of collection is 100% of the application's run-time, they might decide that the impact of collecting the data is too high to warrant collection.

Existing Instrumentation Solutions

To collect data about an application, we instrument the application executable so that when it runs it generates the desired information. Program instrumentation techniques vary from completely manual (programmer inserted directives) to completely automatic (no direct user involvement). Instrumentation can be inserted into the application source code directly, automatically by the compiler, placed into runtime libraries, or even by modifying the linked executable. Each of these approaches has advantages and disadvantages.

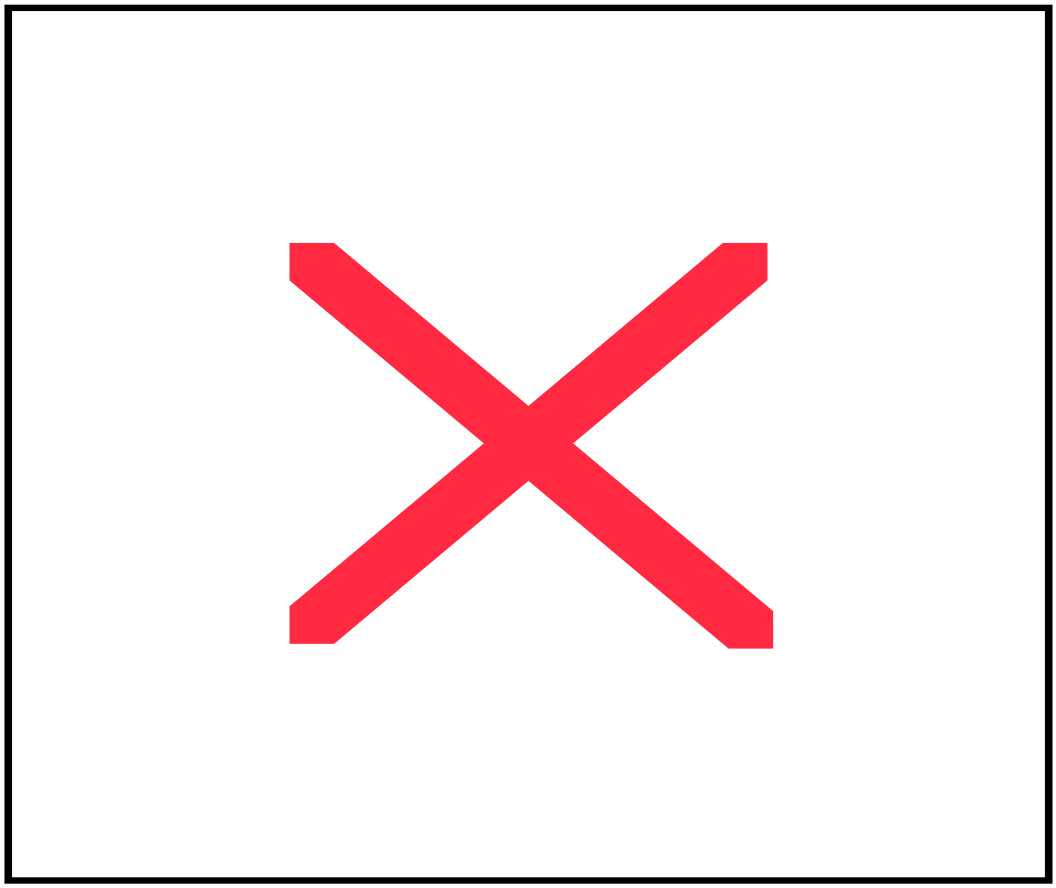

Figure 2: Program Instrumentation can be done at various levels from source code thorough a running program

Figure 2 shows the stages of a compilation and a sample of the places that different tools insert their instrumentation. For example, SvPablo [13] and Prophesy [14] instrument source code automatically, this approach provides maximum portably of source code, but requires tools for each source language. Another approach is to instrument runtime library calls. For example, the MPI

tracing interface provides a way for a tool to intercept all calls to MPI routines. When an MPI program is compiled using this interface, all calls to the MPI library are routed through a "name-shifted" version of each function. Tools are written as a library of routines that record the desired information for each MPI function. This approach provides easy instrumentation in multiple languages, but limits the instrumentation points to library calls. NetLogger [15], ARM [16], and log4j [17] provide routines that application programmers can call to monitor their program. This approach provides maximum flexibility, but requires programmers to hand modify their program to be measured. The Dyninst system [18] permits instrumentation to be inserted into running binaries.

For very detail performance analysis, sometimes it is necessary to be able to see inside of the host operating system. For example, Web100 exposes the statistics inside the TCP stack itself through an enhanced standard "Management Information Base" (MIB) for TCP [19]. If a network-based application is performing poorly, TCP information from Web100 allows one to determine if the bottleneck is in the sender, the receiver, or the network itself. Another project, Monitoring Apparatus for General kerNel Event Tracing (MAGNET) [20] provides a high-fidelity, low-overhead monitoring mechanism for exporting kernel events to user space. The kerninst project [21] provides the ability for user program to inject new code into arbitrary points in the kernel to gather statistics.

To illustrate the tradeoffs in different styles of instrumentation, we briefly present NetLogger, ARM, and Dyninst here.

NetLogger Toolkit

Researchers at Lawrence Berkeley National Lab have been developing a toolkit for instrumenting distributed applications called NetLogger [15]. Using NetLogger, distributed application components are modified to produce timestamped traces of "interesting" events at all the critical points of the distributed system. Events from each component are correlated, allowing one to characterize the performance of all aspects of the system and network in detail.

All the tools in the NetLogger Toolkit share a common monitoring event format, and assume the existence of accurate and synchronized system clocks. The NetLogger Toolkit itself consists of four components: an API and library of functions to simplify the generation of application-level event logs, a service to collect and merge monitoring from multiple remote sources, a monitoring event archive system, and a tool for visualization and analysis of the log files. In order to instrument an application to produce event logs, the application developer inserts calls to the NetLogger API at all the critical points in the code, then links the application with the NetLogger library.

NetLogger events can be formatted as an easy to read and parse ASCII format. To address the overhead issues discussed above, NetLogger includes a highly efficient self-describing binary wire format, capable of handling over 600,000 events per second [22]. NetLogger also includes a remote activation mechanism, and reliability support.

The NetLogger Reliability API provides fault-tolerance features that are essential in Grid environments. For distributed monitoring, a particular challenge is that temporary failures of the network between the component being monitored and the component collecting the monitoring data are relatively common, especially when several sites are involved. The NetLogger API includes the ability to specify a "backup", i.e. fail-over, destination to use. This may be any valid NetLogger destination, but typically is a file on local disk. If the primary destination fails, all data is transparently logged to the backup destination. Periodically, the library checks whether the original destination has "come back up". If so, the library reconnects and, if the backup destination was a file, send over all the data logged during the failure.

The NetLogger Toolkit also includes a data analysis component. One of the major contributions of NetLogger was the concept of linking a set of events together and representing them visually as a "lifeline", as shown in . Visualizing event traces in this manner makes it easy to see where the most time is being spent.

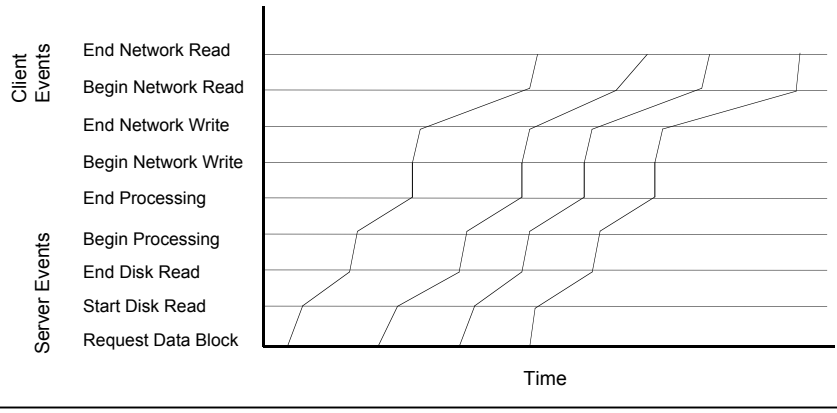

Figure 3: NetLogger Lifelines

The NetLogger Visualization tool, *nlv*, provides an interactive graphical representation of system-level and application-level events. NetLogger's ability to correlate detailed application instrumentation data with host and network monitoring data has proven to be a very useful tuning and debugging technique for distributed application developers.

Application Response Measurement (ARM) and Application Instrumentation and Control (AIC)

The Open Group's Enterprise Management Forum [23] has two products related to instrumentation.

The Application Response Measurement (ARM) API defines function calls that can be used to instrument an application for transaction monitoring. It provides a way to monitor business transactions, by embedding simple calls in the software that can be captured by an agent supporting the ARM API. ARM was originally made available in 1996 by Hewlett-Packard and Tivoli.

The Application Instrumentation and Control (AIC) API provides the facility to instrument an application so that a user of that application will receive real-time information on the progress of work flowing through that application. It also enables the user to exercise control over the work being processed by that application.

Dynamic Instrumentation (dyninst)

Dynamic Instrumentation (dyninst) is the process of adding, removing, and changing instrumentation in a running program [18]. With dynamic instrumentation, data collection probes are inserted into a program while it is running, applying the changes directly to the executing program. Using this technology, decisions as to what kind of instrumentation to use can be deferred until the moment when it is needed, and changes to this instrumentation can be made at any time.

In practice, dynamic instrumentation has been a big success. It pushes the technology to its most flexible limits, with many traditionally cumbersome activities becoming simple. Dynamic instrumentation offers several substantial benefits:

- *Allows changing instrumentation during programs execution*: Initially data is collected about high-level, summary characteristics, then the instrumentation is adjusted to focus on the details of apparent problems. This follows the natural way in which programmers work, but avoids the need to re-run lengthy computations.
- *Allows measurement of large, production-sized applications*: Since instrumentation is inserted dynamically, it can (at run- time) systematically work through the critical parts of the program and skip unimportant ones.

- *Avoids recompiling*: Since changing data collection is much easier than re-compiling, the development process is sped up.
- *Easily integrates new sources of performance data*: Any performance measure that is available from a program's address space easily can be turned into a new metric that can be analyzed and displayed by Paradyn. For example, hardware performance counters that can be mapped into a programmers address space can be easily used.

The features of dynamic instrumentation are available to tool builders via a C++ API [18]. Since the API is the same for all of the supported platforms (currently Solaris/Sparc, x86/Linux, x86/Windows, Power/AIX, MIPS/IRIX, and Alpha/TruUnix64) a Grid monitoring system can be built that has the ability to perform detailed instrumentation no matter where the application is running. Dyninst technology can be used directly via Dyninst[1] or as part of IBM's DPCL Interface.

Sensors and Sensor Management

As Grids are becoming bigger and more complex, there are more components to monitor and manage, and the amount of monitoring data produced by this effort is becoming overwhelming. Some components require constant monitoring, while others only need to be monitored on demand. Sensors for measuring hosts, networks, and services are required.

An approach to active monitoring specifically designed for grid-style computing is the Network Weather Service (NWS) [24]. NWS uses a collection of monitoring servers that measure network latency and bandwidth between pairs of nodes. These observations are then used to predict future network performance. NWS contains a family of predictors, and dynamically selects a prediction function based on the throughput and latency.

An important open issue in network prediction for large-scale computational grids is the time horizon of a prediction. For a prediction to be useful, it must be issued far enough in advance to permit the system to take appropriate action. If the intended system action is static process placement, the forecast must cover the entire lifetime of the process which could be several hours. If the goal is to permit process migration, the forecast must be issued in time to permit an orderly migration to take place.

The RMON protocol [25] provides a way for hubs and switches to record and report statistics about the traffic passing through them. The RMON protocol defines a set of Simple Network Management Protocol (SNMP) Management Information Base (MIB) variables that can be extracted and RMON permits gathering statistics about packet counts, a matrix of traffic by sending and receiving host, and statistics about selected TCP and UDP ports. Several commercial products are able to use RMON to provide performance information for a network. For example, Visualware's VisualProfile [26] provides a Java-based interface that allows monitoring and traffic measurement.

Ganglia [27] is a monitoring package designed to provide information about the status of nodes in a cluster or grid. Nodes running ganglia monitor variables, such as number of CPUs on a node or 1 minute load average, that are useful for scheduling and monitoring clusters. The key features of this system are a simple monitoring process that is run on each node in the system, and an efficient protocol that ensures that timely information is made available without causing excessive data transmission. To do this, Ganglia maintains meta data for each monitored variable with a minimum update interval, and a maximum change threshold. Information is propagated only when the monitored variable has changed by more than the threshold, or if no message has been sent for the required time interval. The Nagios system[28] provides similar monitoring to the

[1] Dyninst is available for download at www.dyninst.org.

Ganglia system. Nagios is primarily aimed at system administration, and includes features to monitor UNIX services, such as SMTP or HTTP, running on each node. The Condor [29] system includes similar monitoring to decide which nodes are available to run jobs.

Figure 4 shows the top-level screen for Ganglia running at the San Diego Super Computer Center. The Grid summary (of 17 clusters) shows the aggregate status of 487 hosts (with 1010 total CPUs). The top left graph shows the status of the processors on the clusters over approximately a one-hour time interval. The top right graph shows the memory utilization for the same period. The other graphs show the same metrics plotted for the individual clusters (due to space limitations, only two individual clusters are shown in the Figure).

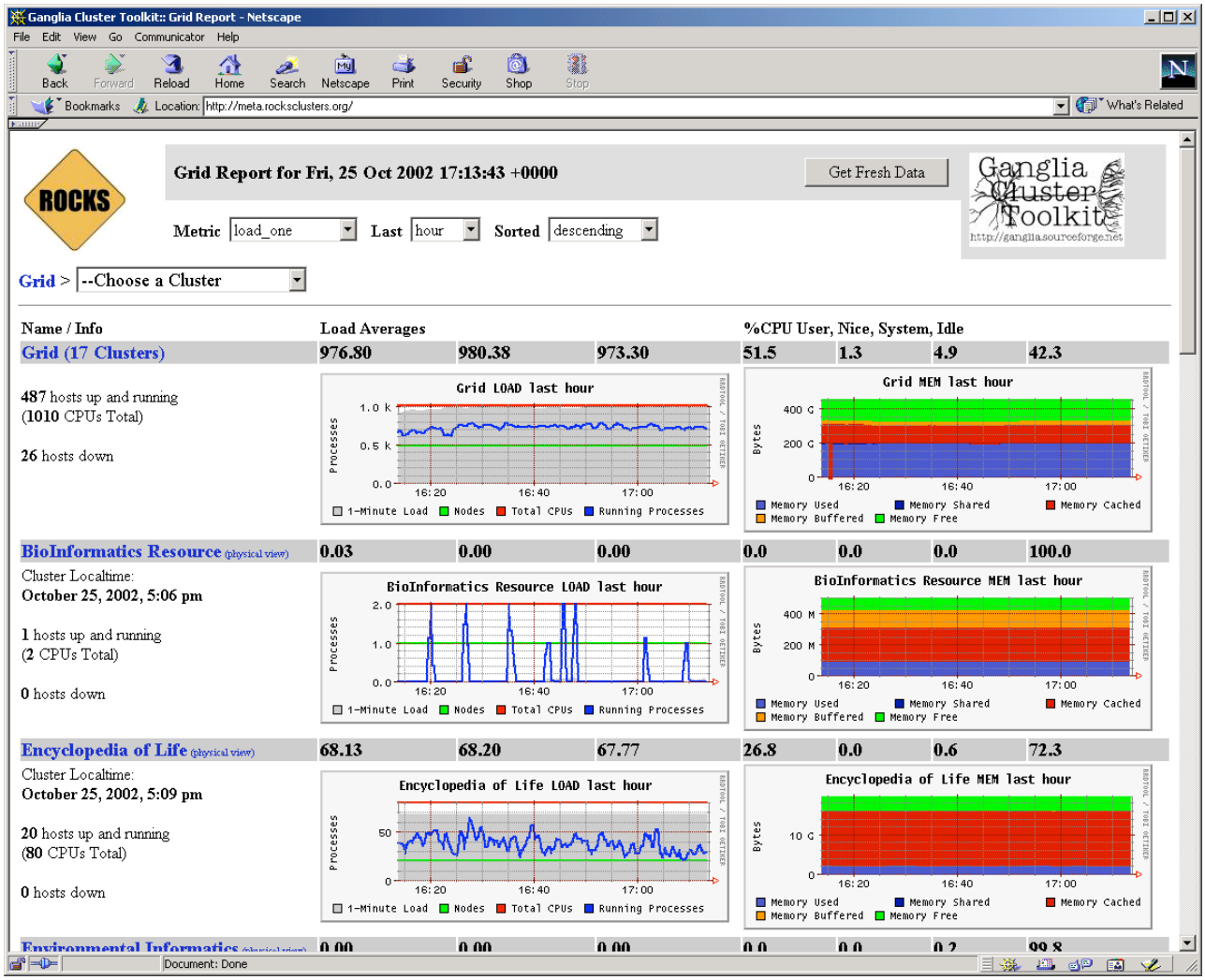

Figure 4: Sample Ganglia Output.

<Need to obtain permission for this figure>

To be useful in a Grid context, monitoring data from tools such as NWS, RMON, and Ganglia must be make available to Grid application and middleware via a monitoring data publication services, as described in the next section.

Information and Monitoring Event Data Publication

In the Grid, a substantial amount of monitoring data must be collected for a variety of tasks such as fault detection, performance analysis, performance tuning, performance prediction, and scheduling. At one site, this data may often be centralized, but in a Grid, the data itself is distributed. Therefore,

to make use of monitoring event data, consumers must first use a monitoring data publication system to locate an appropriate data provider. To publish and interpret monitoring event data, standard schemas and ontologies are required.

Designing such a monitoring-data publication system is challenging due to the diversity, size, dynamic behavior, and geographical distribution of the monitored Grid components that may interest a single monitoring data event consumer.

Requirements

In a Grid environment, the monitoring service must also be distributed. In fact, a Grid would not typically have one monitoring service, but several, unified through a common discovery interface. A Grid monitoring service must be:

- Reliable: If one component is down, the rest of the monitoring system must be useable. All components must be distributed.
- Timely: The monitoring data must be "fresh" enough to be useful. Therefore, the latencies required to both locate and access the monitoring data must be bounded appropriately for a given class of monitoring data.
- Scalable: The system must scale to required levels in terms of both number of producers and consumers of monitoring data. Monitoring event rates may vary widely: events may be generated at rates from several thousand per second (e.g., application instrumentation data) to a few times per year (e.g., a "disk full" error message).
- Secure: Only authorized users or groups should be able to access certain types of monitoring data. Different sites and virtual organization policies must be accommodated.

We now described some examples of existing work in this area: Common Information Model (CIM), Grid Monitoring Architecture (GMA), Open Grid Services Architecture (OGSA) monitoring services, the Monitoring and Discovery Service (MDS), and Relational GMA (R-GMA). GMA and OGSA are general architectural models, MDS and R-GMA are actual working systems, and CIM is both a data model and a publication system. Not discussed are a number of other Grid monitoring systems under development including CODE [30] and Hawkeye [31]. We note that there are many other existing event publication systems, including the event handling components of CORBA and JINI. However, none of the other systems was specifically designed for the requirements of Grid environments.

Common Information Model (CIM)

Many of the monitoring issues discussed here are of course not unique to Grid environments. The Desktop Management Task Force (DMTF), a consortium of several major industry partners, has a number of initiatives in this area, including Common Information Model (CIM), Desktop Management Interface (DMI), Directory Enabled Network (DEN) Initiative, Web-Based Enterprise Management (WBEM), and the Alert Standard Format (ASF) [32].

The Common Information Model (CIM) is an implementation-neutral schema for describing overall management information in a network/enterprise environment [33]. CIM paradigms draw from traditional set theory and classification theory. The schemas can be described using UML.

Besides schemas, the CIM specification also includes a monitoring event publication system. Monitoring events are produced by CIM *providers*. The CIM *Object Manager* (CIMOM) is used to publish the monitoring data and to keep track of who has subscribed to specific events and forwards them to subscribers. Consumers of monitoring data are called *client*s, which can request

or search for data stored in the CIMOM's repository, and subscribe to be notified of changes to any resource known to the CIMOM. An open source version, called *Pegasus* [34], is being developed.

Grid Monitoring Architecture (GMA)

In 1999 the Global Grid Forum (GGF) undertook an effort to define a highly scalable architecture for Grid monitoring, called the Grid Monitoring Architecture, or GMA.

In the GMA, the basic unit of monitoring data is called an event. An event is a named, timestamped, structure that may contain one or more items of data. This data may relate to one or more resources such as memory or network usage, or be application-specific data like the amount of time it took to multiply two matrices. The component that makes the event data available is called a producer, and a component that requests or accepts event data is called a consumer. A directory service is used to publish what event data is available and which producer to contact to request it. These components are shown in Figure 5.

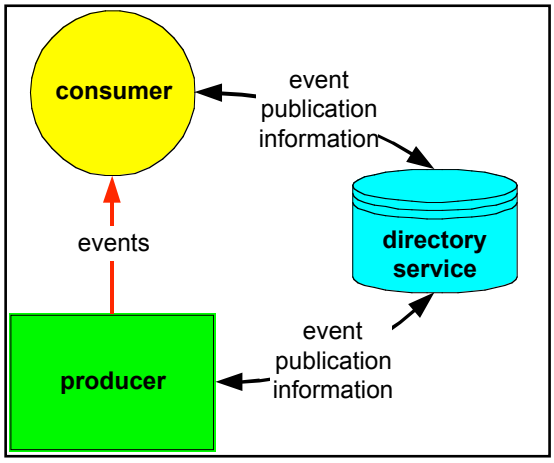

Figure 5: Grid Monitoring Architecture Components

The GMA architecture supports both a subscription model and a request/response model. In the former case, event data is streamed over a persistent "channel" that is established with an initial request. In the latter case, one item of event data is returned per request. The unique feature of GMA is that performance monitoring data travels directly from the producers of the data to the consumers of the data.

A producer and consumer can be combined to make what is called a producer/consumer pipe. This can be used, for example, to filter or aggregate data. For example, a consumer might collect event data from several producers, and then use that data to generate a new derived event data type, which is then made available to other consumers. More elaborate filtering, forwarding, and caching behaviors could be implemented by connecting multiple consumer/producer pipes.

Although the GMA is general-purpose, However, it omits many details necessary to build interoperable monitoring systems. A number of groups have are now developing monitoring services that are based on this architecture, such as REMOS [35] and TOPOMON [36].

Monitoring and Discovery Service (MDS)

The Monitoring and Discovery Service (MDS) is the information services component of the Globus Toolkit™. The current Monitoring and Discovery Service (MDS), version 2, consists of two components, a configurable information provider component called a Grid Resource Information Service (GRIS) and a configurable aggregate directory component called a Grid Index Information Service (GIIS).

MDS provides the necessary tools to build an LDAP-based information infrastructure. MDS uses the LDAP protocol as a uniform means of querying system information from a variety of system components, and for optionally constructing a uniform namespace for resource information across a system that may involve many organizations.

The GRIS is a distributed service that can answer queries about a particular resource by directing the query to an information provider for a given grid resource, and is responsible for authenticating incoming information requests. The GIIS provides a means of combining arbitrary

GRIS services to provide a coherent system image that can be explored or searched by grid applications.

In MDS, interactions between higher-level services (or users) and providers are defined in terms of two basic protocols: a soft-state *registration protocol* for identifying entities participating in the information service, and an *enquiry protocol* for retrieval of information about those entities.

Relational GMA (R-GMA)

Another Grid monitoring and information service is R-GMA [37]. Its functionality is similar to the MDS, but where MDS is based on LDAP, R-GMA is based on a relational model with SQL support. LDAP has the benefits of being easy to distribute and having well defined protocols, but the data model is rather inflexible and the query language supports only simple queries. Because of this, the R-GMA approach is to use a relational model. The relational model organizes data in tables. Representing monitoring data in this way allows for a much more powerful query ability, but has the drawback that it is difficult to do distributed relational queries. However mechanisms for performing relational queries on distributed databases is a very active research area, and products are starting to emerge that support this.

Open Grid Services Architecture

The Open Grid Services Architecture (OGSA), described in detail in chapter N, has been designed to support the functionality required to implement a monitoring event publication service.

The OGSA *notification framework* allows clients to register interest in being notified of particular messages and supports asynchronous, one-way delivery of such notifications. If a particular service wishes to support subscription of notification messages, it must support the *NotificationSource* interface to manage the subscriptions. A service that receives notification messages implements the *NotificationSink* interface, which is used to deliver notification messages. The *Service Data Element* is used to describe the type and format of the data.

Most of the ideas of the OGSA notification framework have a direct mapping to GMA concepts. The GMA Producer and Consumer are roughly equivalent to the OGSA *notification source* and *notification sink*. The GMA Event is equivalent to the OGSA *notification message*, and the GMA directory service is similar to the OGSA *registry*. A new version of the MDS is being developed that is based on OGSA.

Other Issues

Several open issues must be resolved to achieve the goal of uniform access to Grid monitoring data. CIM work is addressing the issue of common schemas, but there are several Grid components that are not yet part of the CIM schemas. These include batch schedulers, replica managers, and metrics for network paths. A Global Grid Forum working group is attempting to defining missing CIM schemas for the Grid. Emerging web services technologies such as OGSA, WSDL, SOAP, XSchema, and UDDI may meet some of the challenges of Grid monitoring interoperability. However, a great deal of work remains to be done to define everything necessary to produce interoperable Grid monitoring services, including common event data descriptions, event dictionaries, and query formats. Several working groups in the Global Grid Forum are addressing these issues as well; incorporating the best ideas from all the systems described here.

Monitoring Event Data Archives

Monitoring event data archives are critical for performance analysis, tuning, and accounting. In a Grid environment, conditions over which users have no control are constantly changing. Historical data can be used to establish a baseline for analysis of current performance.

As an example, a user of a Grid File Replication service [38] notices that generating new replicas is taking much longer than it did last week. The user has no idea why performance has changed -- is it the network, disk, end host, GridFTP server, GridFTP client, or some other Grid middleware such as the authentication or authorization system?

To determine what changed, the user needs to analyze monitoring data from hosts (CPU, memory, disk), networks (bandwidth, latency, route), and the FTP client and server programs. Depending upon the application and systems being analyzed, from days to months of historical data may be needed. Sudden changes in performance might be correlated to other recent changes, or may turn out to occur periodically in the past. In order to spot trends and interactions, the user needs to be able to view the entire dataset from many different perspectives.

Archive Requirements

A monitoring archive aggregates monitoring events from different tools, across different systems, over a long period of time. Many useful queries across this kind of dataset will be to find relations between the characteristics of these events. Therefore, monitoring data archives should support relational queries.

Although different monitoring data archives will have different schemas, some basic characteristics of monitoring events should be preserved by any archive. Each archived monitoring event needs at least a precision timestamp, a global name for the type of monitoring event, an identifier for the "target" of that event, and one or more associated values. The global name is useful because an archive is only one possible place to store a monitoring event, and the event name serves to tie together archives, memory caches, flat files, analysis programs, etc. in an implementation-neutral way.

Monitoring archives can, in general, afford to trade off response time for (data update) throughput. Fast response times are usually required only for recent or averaged data, and this problem is best solved with some sort of monitoring data cache. But aggregate monitoring data may arrive at high rates for an extended period of time, so if the system throughput is too low, it may fall behind and data in the archive will always be stale.

Given these potentially high monitoring event rates, it is important to ensure that the act of sending monitoring event data to the archive is not intrusive. Monitoring event data should be buffered on disk, and sent to the archive slowly in the background. This implies that there must be sufficient buffer space within Grid nodes to prevent data loss during transient periods of high event frequency.

Example Monitoring Data Archive Systems

The Prophesy project includes a monitoring archive database [14]. Prophesy collects detailed pre-defined monitoring information at a function call level. The data is summarized in memory and sent to the archive when the program completes. In addition, Prophesy includes modeling and prediction components.

Another system that provides access to historical data is PperfDB [39], which provides a way to store performance measurements for different program executions. The system also

includes an automatic difference operator that allows programmers to discover what changed between two different executions of a program.

NetLogger includes a relational monitoring event archive, which provides the ability to correlate events and understand performance problems in distributed systems. The NetLogger project has shown how a relational database with historical data allows for the establishment of a baseline of performance in a distributed environment, and finding events that deviate from this baseline is easy with SQL queries.

Data Analysis

Collecting monitoring data is only the start of the process. To be useful, data must be analyzed. Several other chapters of this book describe how gathered data can be used by various parts of Grid middleware. For example, Condor (Chapter X) and Autopilot (Chapter X) use monitoring data to make decisions about how to run programs. Those chapters concentrate on how gathered data is used by the system software to help operate the Grid. In this section, we concentrate on how performance data that has been gathered by a running program can be used to identify bottlenecks in the application, or to re-configure the application to better utilize the available Grid resources.

Automatic Performance Diagnosis

Paradyn's Performance Consultant (PC) [3] can automatically control the dynamic instrumentation. The PC has a well-defined notion of performance bottlenecks, based on identifying the parts of the program that are consuming the most resources, and then categorizing the type of problem (e.g., synchronization, I/O, memory, and CPU) causing the bottleneck. Using fairly simple heuristics, the PC is able to guide the programmer to the source of many performance problems. The Paradyn system is designed for identifying bottlenecks in parallel program written using message passing or pthreads.

Combining the PC and dynamic instrumentation allows Paradyn to automatically control the level of instrumentation overhead. It uses a feedback-based model to control the cost of instrumentation. The application programmer sets the threshold (percent of CPU use) for Paradyn's instrumentation, and the PC automatically controls its rate of searching for bottlenecks based on this value

The programmer simply tells the PC to start searching for performance problems. The PC will select parts of the program to instrument, evaluate the results from that instrumentation, and then try to refine its instrumentation based on the results. Figure shows a search in progress. The blue nodes are types of problems or program parts that have been currently identified as bottlenecks, the underlined nodes are ones that have been rejected as bottlenecks, and the rest currently are being evaluated. If you follow the highlighted nodes, one of the problems that the PC found is that there is a “ExcessiveSyncWaitingTime” (i.e., the time spending waiting on synchronization is above the user-specified threshold). This synchronization problem has been localized to module (file) s_msg.o, and further localized to procedure s_recv.

The PC gives the programmer a strong head start in trying to locate performance problems; in many cases, the PC provides sufficient information to completely locate a problem. Once a problem is shown to the programmer, they can use the Paradyn visualizations to quantify the nature of the problem.

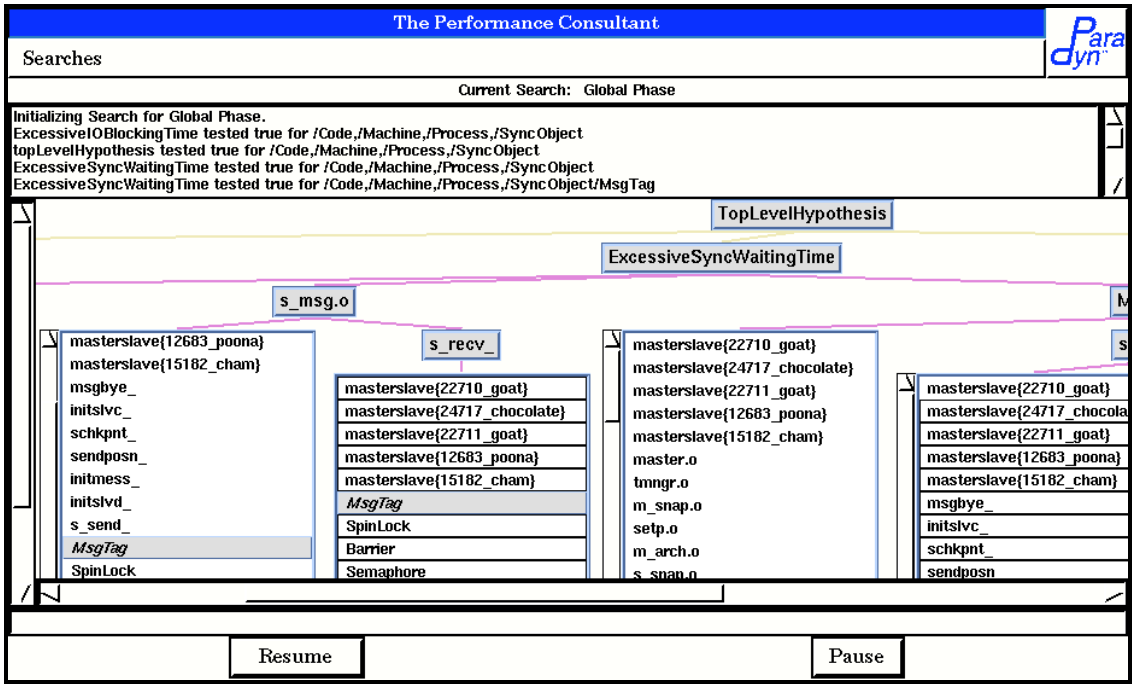

Figure 7: Paradyn Automatic Performance Search.

Automatic Summarization

The Prophesy system [40] and SvPablo [13] provide a way to capture summary performance data (rather than event logs) from grid applications to be used for data analysis. The idea is that rather than recording events of everything that happened during the program's execution that the tool only records the summary information for specific activities. For example, the Prophesy tool records information about communication points (sends and receives), subroutines and loops, and I/O. For each point (e.g. a call to a communication routine or a loop head), the system records information about the frequency and duration of the activity. At the end of the program's execution, data is saved about each point.

Automatic Adaptation

One of the most promising areas of using monitoring data for Grid applications is in the area of automatic program adaptation. Since applications are no longer monolithic programs written for a specific purpose. Instead, most software today makes extensive use of libraries and re-usable components. This approach generally results in software that is faster to build and more modular. However, one problem with this approach is that the various libraries used by an application are not tuned to the specific application's need. In addition, applications are frequently used in very different ways. For example, different users may employ a single commercial simulation application for radically different types of simulations. Because of this reuse of software, applications may not run well in all configurations.

The transient, rarely repeatable behavior of Grid computing environment indicates the need to replace standard models of post-mortem performance optimization with a real-time model, one that optimizes application and runtime behavior during program execution. Automatic program library selection provides a framework to help with this goal; it helps to tune the application during runtime execution by monitoring the underlying library performance and switching underlying program library as needed. This is an important step toward automated performance tuning in the Grid computing.

Two systems that have tried to provide this type of flexibility are AppLeS [41] and Active Harmony [42]. In the AppLeS system, a toolkit is provided to allow writing applications that are aware of the currently available resources. In the Active Harmony system (described in the case study below), applications expose parameters about different configurations and algorithms they support. A runtime layer then monitors application performance and Grid parameters and adjusts the tunable parameters for the harmonized application.

Case Studies

In this section, we present two case studies using instrumentation and monitoring tools to improve the performance of Grid applications. The first case study shows *Active Harmony* used to dynamically tune a Grid application. The second case study presents a complete end-to-end instrumentation and monitoring use case combining a number of components to collect data for analysis using the NetLogger performance analysis tool.

Dynamic Tuning with Active Harmony

Active Harmony is an infrastructure that allows applications to become tunable by applying minimal changes to the application and library source code. This adaptability provides applications with a way to improve performance during a single execution based on the observed performance. The types of things that can be tuned at runtime range from parameters such as the size of a read-ahead parameter to what algorithm is being used (e.g., heap sort vs. quick-sort).

Active Harmony demonstrates the use of instrumentation, event publication, and runtime data analysis. In Active Harmony, applications are instrumented to report their performance characteristics and available tuning options. Each program component (library, or different coupled applications) then publishes this information to the Harmony server that contains the data analysis modules. The Harmony server has no information about specific applications; instead it is an optimization framework that is able to running any application that can publish monitoring data and tunable parameters.

To illustrate the techniques, consider a 3-d volume reconstruction application [43] built on top of the Active Data Repository (ADR) middleware [44]. The 3-d volume reconstruction application uses digital images of a space to reconstruct the objects that are visible from the various camera angles. The ADR is an infrastructure that integrates storage, retrieval and processing of large multi-dimensional data sets. ADR provides the user with operations including index generation, data retrieval, scheduling across parallel machines, and memory management. The data is accessed through range queries (i.e., extract all data within a specified region of space). A range query is processed in two steps: query planning followed by query execution. As part of query execution, input and output items are mapped between coordinate systems and the data is aggregated to generate the final result. During the processing phase, a temporary dataset called the accumulator, is created to hold the results of the query being processed.

Because ADR is middleware used to build multiple applications, including the Harmony calls in the ADR code makes every application built on top of ADR tunable. The parameters used were:

- **tileSize** represents the size of the memory tile that is used by the ADR back-end to store information before it is aggregated. It is the size of the tiles that the accumulator will be partitioned if it does not fit into memory. This parameter greatly influences query planning and execution since it is somewhat analogous to the block size in a computational code that has been blocked (tiled) to fit into a cache.

- **lowWatermark** is the upper bound of the number of pending reads and number of ready reads that were issued to the disk in order to resolve a certain query.
- **maxReads** is the maximum number of reads issued in order to resolve the current query if the number of pending read operations and the number of ready read operations are below the lowWatermark.

The original version of the volume reconstruction application used values for the parameters provided by the ADR designers. The application was harmonized by adding calls to expose these parameters to the system. The experiments were run on a Linux cluster with 16 machines, each with two 450 MHz processors connected by 100 Mbps Ethernet.

Figure 8 below presents the improvement obtained in the processing time of each of the queries. The Active Harmony system sped up query processing by up to 50% for the set of 70 random queries that we generated. However, the average improvement was about 10%. This is due to the fact that some of the queries that had the greatest speed-ups were very short, compared with others for which the improvement was less than 10%. The performance improvement for the longest query, which took about 10 minutes to be completed, was about 18%.

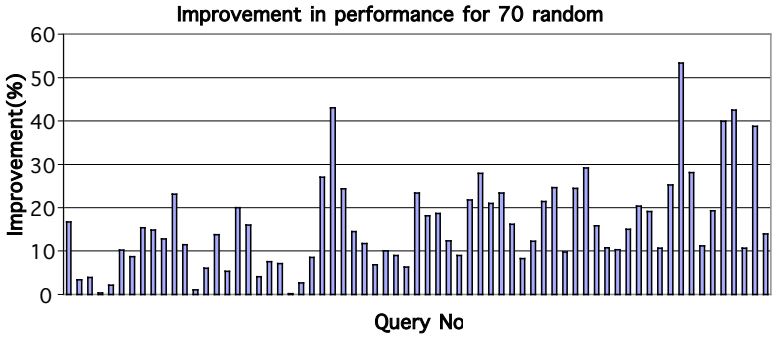

Figure 8: Performance improvement for the volume reconstruction application.

One interesting thing about Active Harmony is its ability to optimize an application when shape of the performance curve, and thus what the best value is, is unknown prior to execution. This is accomplished by using heuristic search techniques. Figure 9 below shows how this works. In a Grid environment, the ability to optimize a program without detailed knowledge of the full objective function is critical since components of an application may never have been used together before and (due to Grid schedulers) may never be used together again.

In this illustration, the same query for each tuple of parameter values submitted to the ADR back-end with different values for the three application parameters. By sweeping through all parameter combinations, it is possible to construct a complete performance curve. The complete parameter sweep recorded values for the performance function of up to 25% slower than the optimum, while the range of values explored by the Active Harmony system was within 5% of the minimum. The minimum was reached by exploring only 11 tuples (out of the almost 1,700 different possible tuples).

The axes of the graph in Figure 9 are as follows: the vertical one represents the performance function, while the horizontal ones are the tileSize and the lowWatermark. The values obtained for different values of the third parameter: maxReads are stacked one on top of the other in the graph. The lighter points in the graph are from the exhaustive search and are spread on the entire value space. The darker points (lower left corner) trace the path followed by the tuning and they are concentrated near the minimum.

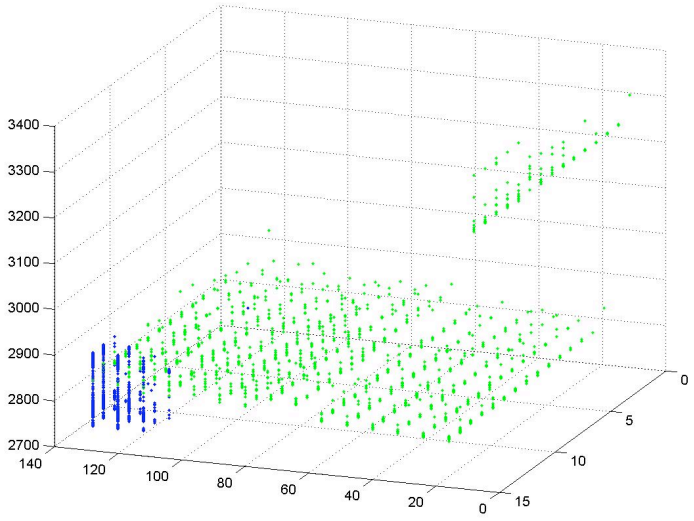

Figure 9: Performance curve (via exhaustive search) for the volume reconstruction application.

End-to-End Monitoring Example

In this section, we present an end-to-end instrumentation and monitoring example. This example includes all the aforementioned monitoring components: instrumentation, monitoring event publication service, monitoring event archive, sensor management systems, and data analysis.

Consider the problem of developing, tuning, and running a reliable bulk data transfer service for the Grid. The first step is to insert instrumentation code during the development stage to ensure the program is operating as expected. This can be done using an instrumentation package such as ARM or NetLogger, and instrumentation code should be added to generate timestamped monitoring events before and after all disk and network I/O. Most modern bulk data transfer programs, such as *GridFTP* [45] and *bbftp* [46], are multithreaded and support parallel data streams. Threaded programs such as this can often have hidden bugs, and instrumentation is important to ensure the program is behaving properly.

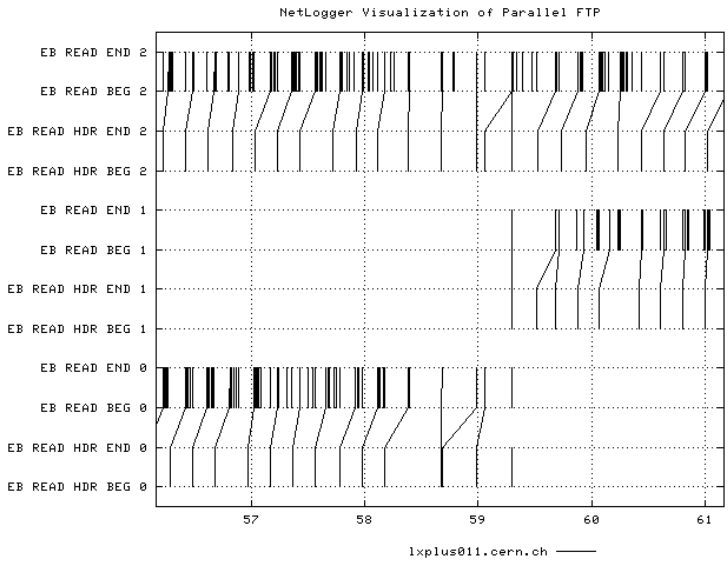

Figure 10: NetLogger Visualization for Parallel FTP debugging.

An example of this is shown in Figure 10, which shows NetLogger lifelines for the data receiver. The three groups of lifelines are traces for each of three separate read sockets, and includes monitoring events for reading header and data packets. All three streams should always be active, but as shown in the graph, streams zero and one would start and stop somewhat randomly. The cause of this was a simple logic bug that selected the wrong socket for reading when data was available on multiple sockets. This type of subtle bug could easily remain undetected without this type of analysis and visualization.

Now that the file transfer service is debugged and tested, it is ready to put into production use on the Grid. This is where the remaining monitoring services come into use.

The level of instrumentation required for the debugging scenario above can easily generate thousands of monitoring events per second. Clearly one does not need or want this level of monitoring activated all the time, so some type of monitoring activation service is needed, which allows a user to turn instrumentation on and off in a running service.

Next, it is useful to establish a performance baseline for this service, and store this information in the monitoring event archive. System information such as processor type and speed, OS version, CPU load, disk load, and network load data should be collected during the baseline test runs. Network monitoring data, such as network capacity, available bandwidth, delay, and loss, should also be collected. The sensor management service again may be needed to start system and network sensors before running the baseline tests. If sensors are already running, the monitoring event publication service is needed to locate the sensors and initiate a subscription for the resulting monitoring data. Several tests are then run, sending complete application instrumentation (for clients, servers, and middleware), host, and network monitoring data to the archive. It is probably not possible to establish baseline for all possible network paths that may be used, but it is useful to try to collect baseline data for several paths that represent typical paths in a given Grid.

This data transfer service is then put into production use, and everything works fine for a couple months. One-day users start complaining that data transfers are taking much longer than they used to. Without end-to-end monitoring and instrumentation, it can be very hard to track down the cause of this. Is it a software, hardware, or network issue?

At this point one needs to collect data for analysis. This requires the following steps:

- Locate relevant monitoring data in the monitoring event publication service and subscribe to that data.
- Activate any missing sensors using the sensor management system, and subscribe to the data they produce.
- Activate instrumentation of the data transfer service, and subscribe to that data.
- Locate monitoring data in the monitoring event archive for the baseline test from when things were last working.

Now one can begin the data analysis. This could include the following steps:

- Check the hardware and OS information to see if anything changed.
- Look at the application instrumentation data to see if anything looks unusual.
- Look at the system monitoring data to see of anything looks unusual (e.g.: unusually high CPU load).
- Correlate the application and middleware instrumentation data with the host and network monitoring data.

As an example, see Figure 11. For this figure, we used the NetLogger visualization tool, *nlv*, to correlate client and server instrumentation data with CPU and TCP retransmission monitoring

data. The events being monitored are shown on the y-axis, and time is on the x-axis. From bottom to top, one can see CPU utilization events, application events, and TCP retransmit events all on the same graph. Each semi-vertical line represents the "life" of one block of data as it moves through the application. The gap in the middle of the graph, where only one set of header and data blocks are transferred in three seconds, correlates exactly with a set of TCP retransmit events. Thus, this plot makes it easy to see that the "pause" in the transfer is due to TCP retransmission errors on the network.

This tells us that the change in performance was due to network congestion. Tracking down the cause of the network congestion is a difficult problem, and requires a large amount of network monitoring data. This problem is also facilitated by services such as monitoring event publication system and monitoring event archives. The Internet2 End-to-End Performance Initiative (http://e2epi.internet2.edu/) is addressing this problem.

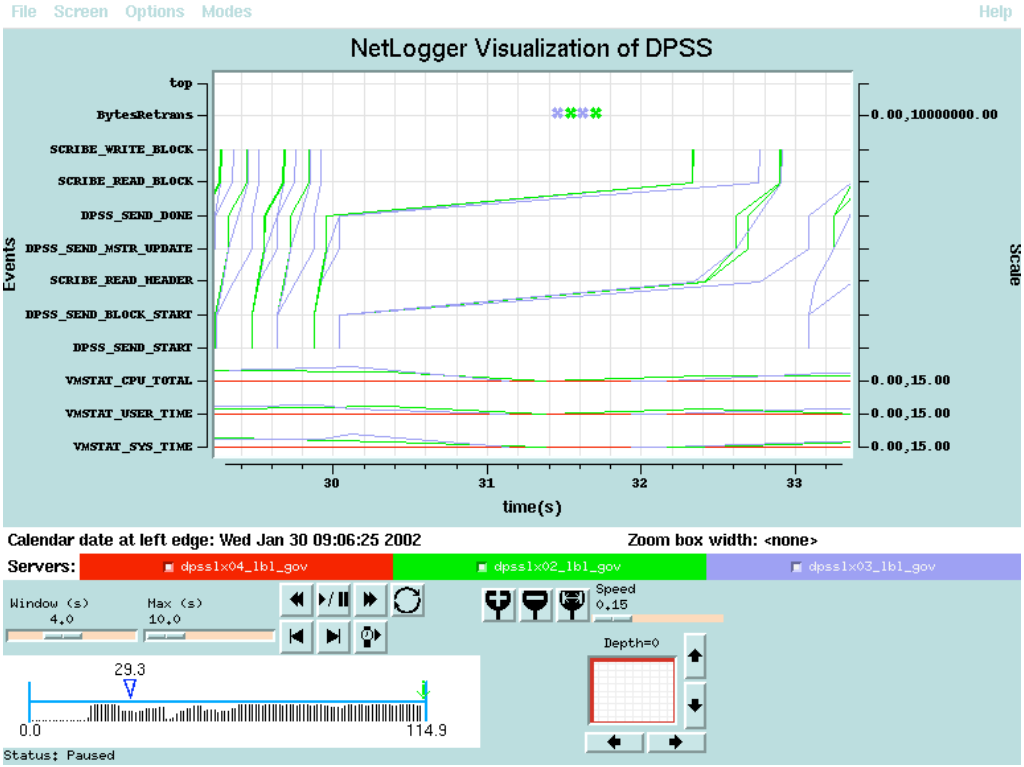

Figure 11: NetLogger Visualization System.

Recommended Reading:

http://www.gridforum.org/1_GIS/GIS.htm

http://www-didc.lbl.gov/GGF-PERF/GMA-WG/

http://www.dmtf.org/standards/standard_cim.php

References

1. Geist, G.A., et al., *PICL - A Portable Instrumented Communication Library*. 1990, Oak Ridge National Laboratory.TM-11130.
2. Reed, D.A., et al., *Scalable Performance Analysis: The Pablo Performance Analysis Environment*, in *Scalable Parallel Libraries Conference*, A. Skjellum, Editor. 1993, IEEE Computer Society. p. 104-113.
3. Miller, B.P., et al., *The Paradyn Parallel Performance Measurement Tools*. IEEE Computer, 1995. **28**(11): p. 37-46.
4. Mathisen, T., *Pentium Secrets*. Byte, 1994. **19**(7): p. 191-192.
5. *DECchip 21064 and DECchip21064A Alpha AXP Microprocessors - Hardware Reference Manual*. 1994, DEC.EC-Q9ZUA-TE.
6. Mills, D.L. *Improved Algorithms for Synchronizing Computer Network Clocks*. in *SIGCOMM*. 1994. London.
7. Lamport, L., *Time, Clocks, and the Ordering of Events in a Distributed System*. CACM, 1978. **21**(7): p. 558-564.
8. Browne, S., et al. *A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters*. in *Proc. SC'2000*. 2000. Dallas, TX.
9. Bernstein, D., A. Bolmarcich, and K. So. *Performance visualization of parallel programs on a shared memory multiprocessor system*. in *International Conference on Parallel Processing (ICPP)*. 1989. University Park, PA, USA.
10. Malony, A., *Performance Observability*. 1990, Department of Computer Science, University of Illinois.
11. Lumpp, J.E., *Models for Recovery from Software Instrumentation Instrusion*, in *Electrical Engineering Department*. 1993, University of Iowa.
12. Hollingsworth, J.K. and B.P. Miller. *An Adaptive Cost Model for Parallel Program Instrumentation*. in *Euro-Par'96*. 1996. Lyon, France: Springer.
13. deRose, L.A. and D.A. Reed. *SvPablo: A Multi-Language Architecture-Independent Performance Analysis System*. in *International Conference on Parallel Processing*. 1999. Wakamatsu, Japan.
14. Wu, X., V. Taylor, and e. al. *Design and Development of Prophesy Performance Database for Distributed Scientific Applications*. in *Proc. the 10th SIAM Conference on Parallel Processing for Scientific Computing*. 2001. Virginia.
15. Tierney, B., et al. *The NetLogger Methodology for High Performance Distributed Systems Performance Analysis*. in *Proc. 7th IEEE Symp. on High Performance Distributed Computing*. 1998.
16. The Open Group. *Application Response Measurement (ARM)*2002, http://www.opengroup.org/management/arm.htm http://www.opengroup.org/management/arm.htm.
17. The Apache Software Foundation. *The Jakarta Project*2003, The Apache Software Foundation http://jakarta.apache.org/log4j/docs/.
18. Buck, B. and J.K. Hollingsworth, *An API for Runtime Code Patching*. Journal of Supercomputing Applications, 2000. **14**(4): p. 317-329.
19. Mathis, M., et al. *TCP Extended Statistics MIB*2002, IETF http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-tcp-mib-extension-02.txt.
20. Gardner, M.K., et al., *MAGNET: A Tool for Debugging, Analysis and Reflection in Computing Systems,*. 2002, Los Alamos National Labratory.Technical Report LA-UR 02-7170.
21. Tamches, A. and B.P. Miller. *Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels*. in *Third Symposium on Operating Systems Design and Implementation (OSDI)*. 1999. New Orleans.
22. Gunter, D., et al. *Dynamic Monitoring of High-Performance Distributed Applications*. in *11th IEEE Symposium on High Performance Distributed Computing*. 2002.
23. Open Group. *Enterprise Management Forum*2002, http://www.opengroup.org/management/arm.htm
24. Wolski, R. *Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service*. in *High Performance Distributed Computing (HPDC)*. 1997. Portland, Oregon: IEEE Press.

25. Waldbusser, S. *Remote Network Monitoring Management Information Base*. in *RFC 1757*. 1995: IETF.
26. Visualware. *VisualProfile Enterprise Edition 2.0*2002, http://www.visualware.com/visualprofile http://www.visualware.com/visualprofile/index.html.
27. Ganglia Development Team. *Ganglia Meta Daemon*2002 http://ganglia.sourceforge.net.
28. Galstad, E. *Nagios*2002, http://www.nagios.org http://www.nagios.org.
29. Litzkow, M., M. Livny, and M. Mutka. *Condor - A Hunter of Idle Workstations*. in *International Conference on Distributed Computing Systems*. 1988.
30. Smith, W., *A Framework for Control and Observation in Distributed Environments*. 2001, NAS Technical Report Number: NAS-01-006
31. Condor Project. *Hawkeye*2002, http://www.cs.wisc.edu/condor/hawkeye/ http://www.cs.wisc.edu/condor/hawkeye/.
32. Desktop Management Task Force. *DMTF*2002, http://www.dmtf.org/standards/index.php http://www.dmtf.org/standards/index.php.
33. Desktop Management Task Force. *Common Information Model (CIM) Standards*2002, http://www.dmtf.org/standards/standard_cim.php http://www.dmtf.org/standards/standard_cim.php.
34. The Open Group. *Pegasus*2002, http://www.opengroup.org/management/pegasus/ http://www.opengroup.org/management/pegasus/.
35. Dinda, P., et al. *The Architecture of the Remos System*. in *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*. 2001. San Francisco, CA: IEEE Computer Society.
36. Burger, M.d., T. Kielmann, and H.E. Bal. *TOPOMON: A Monitoring Tool for Grid Network Topology*. in *International Conference on Computational Science*. 2002.
37. Fisher, S. and WP3, *Information and Monitoring Services Architecture: Design, Requirements and Evaluation Criteria*. 2001
38. Chervenak, A. and e. al. *Giggle: A Framework for Constructing Scalable Replica Location Services*. in *Proceeding of the IEEE Supercomputing 2002*. 2002.
39. Karavanic, K.L. and B.P. Miller. *Experiment Management Support for Performance Tuning*. in *SC'97*. 1997. San Jose, CA.
40. Wu, X., V. Taylor, and R. Stevens. *Design and Implementation of Prophesy Automatic Instrumentation and Data Entry System,*. in *Proc. of the 13th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS2001)*. 2001. Anaheim, CA.
41. Berman, F. and R. Wolski. *Scheduling from the perspective of the application*. in *Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*. 1996. Syracuse, NY, USA 6-9 Aug. 1996.
42. Tapus, C., I.-H. Chung, and J.K. Hollingsworth. *Active Harmony: Towards Automated Performance Tuning*. in *SC'2002*. 2002. Baltimore, MD.
43. Borovikov, E., A. Sussman, and L. Davis. *An Efficient System for Multi-Perspective Imaging and Volumetric Shape Analysis*. in *Workshop on Parallel and Distributed Computing in Image Processing, Video Processing Multimedia (PDIVM'20001)*. 2001: IEEE Computer Society Press.
44. Kurc, T., et al. *Querying Very Large Multi-dimensional Datasets in ADR*. in *Proceedings of SC99*. 1999. Orlando, FL: ACM Press.
45. Allcock, B., et al. *Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing*. in *Mass Storage Conference*. 2001.
46. Babar Project. *bbftp*2002, http://doc.in2p3.fr/bbftp/